Bolt Beranek and Newman Inc.

(12)

Report No. 4817

Development of a Voice Funnel System

Quarterly Technical Report No. 11
1 February 1981 to 30 April 1981

March 1982

Prepared for:
Defense Advanced Research Projects Agency

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO. AD-A112163

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Development of a Voice Funnel System
Quarterly Technical Report No. 11

5. TYPE OF REPORT & PERIOD COVERED
Quarterly Technical
1 February 81-30 April 81

6. PERFORMING ORG. REPORT NUMBER
4817

7. AUTHOR(s)
R. D. Rettberg

8. CONTRACT OR GRANT NUMBER(s)
MDA903-78-C-0356

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Bolt Beranek and Newman Inc.
10 Moulton Street
Cambridge, MA 02238

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
ARPA Order No. 3653

11. CONTROLLING OFFICE NAME AND ADDRESS
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

12. REPORT DATE
March 1982

13. NUMBER OF PAGES
30

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Voice Funnel, Digitized Speech, Packet Switching, Butterfly Switch, Multiprocessor.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This Quarterly Technical Report covers work performed during the period noted on the development of a high-speed interface, called a Voice Funnel, between digitized speech streams and a packet-switching communications network.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Report No. 4817 Bolt Beranek and Newman Inc.

DEVELOPMENT OF A VOICE FUNNEL SYSTEM

QUARTERLY TECHNICAL REPORT NO. 11
1 February 1981 to 30 April 1981

March 1982

This research was sponsored by the
Defense Advanced Research Projects
Agency under ARPA Order No.: 3653
Contract No.: MDA903-78-C-0356
Monitored by DARPA/IPTO
Effective date of contract: 1 September 1978
Contract expiration date: 31 December 1981
Principal investigator: R. D. Rettberg

Prepared for:

Dr. Robert E. Kahn, Director
Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, VA 22209

Table of Contents

1. Introduction

This Quarterly Technical Report, Number 11, describes aspects of our work performed under Contract No. MDA903-78-C-0356 during the period from 1 February 1981 to 31 April 1981. This is the eleventh in a series of Quarterly Technical Reports on the design of a packet speech concentrator, the Voice Funnel.

This report describes the tools that are being used to develop software for the Voice Funnel project. Most of these tools were developed elsewhere and are in widespread use in the research community. However, a new assembler, ASIX, was developed under UNIX* to replace the previous ad-hoc assembler. This assembler deals properly with the span-dependent instructions of the MC68000.

We have included a copy of the current manual pages for these tools as an appendix to this report so that the facilities and options available may be more readily understood. As with most UNIX tools, these manual pages are available in an online form.

* UNIX is a registered trademark of Bell Laboratories.

2. Software Development Tools

The software for the Voice Funnel is being written in the system implementation language "C" and in assembly language for the Motorola 68000. We are using a VAX 11/780 running the Version 4 Berkeley VAX UNIX as the host for software development. We were fortunate that many of the necessary software tools had been developed by groups at Berkeley and MIT and we are grateful for their cooperation and support in providing these tools for our use. We intend to continue this spirit of cooperation by offering current versions of these tools and our enhancements to other members of the research community.

The largest change we have made to these tools was to develop a new assembler which is compatible with the file format of the Version 4 Berkeley VAX tools. This new assembler, which offers better diagnostics and span-dependent instruction processing, is described in Section 2.1 below.

The major pieces of software in the new tools package are the following:

bcc - similar to cc in that it invokes compiler, assembler, linker, ... as needed.

bc68 - a revised version of the C compiler from MIT.

bo68 - a revised version of the C optimizer.

asix - a totally rewritten M68000 assembler. This one is capable of correctly handling span-dependent instructions and uses an output format which is compatible with the Berkeley tools. The assembler will handle symbol names of arbitrary length, as will the linker.

blnk - a version of the VAX loader modified slightly for use with the M68000. It is very fast and provides special facilities such as the ability to specify the starting point of the text, data, and bss segments.

bld - A VAX to Butterfly loader which operates over VAX tty lines and supports loading requests from the Butterfly as well as from the user.

bldprom - a utility which sends object files to a DATA I/O prom programmer.

bmake - a program maintenance tool like "make" which understands the 68000 tools and file extensions.

The Voice Funnel project at BBN has its own manual system which works just like the standard UNIX system, but is invoked with "bman" instead of "man". Copies of the current manual pages for these programs are provided in the appendix of this report.

A library suitable for use with the new tools is located in /usr/butterfly/lib/libbs.a. The bcc command will include this library for you when loading. The sources for this library may be found in /usr/butterfly/lib/bsrc.

In the interest of making 68000 files more easily recognizable, we have adopted the following extensions for file names. These extensions are simply the UNIX system extensions with "68" appended.

.c C language source;

.c68 C language source that is specific to the 68000;

.a68 ASIX (68000 assembler) source file;

.s68 ASIX source file generated by a program rather than a person so that directories can be cleaned up without deleting the fruits of human effort;

.o68 68000 object code files, i.e. an input file to "blnk";

.68 the final "executable" file, i.e. a file that is ready to be loaded into the funnel.

The normal namelist facilities (e.g. "nm") of the system will work on the ".o68" and ".68" files since they are in the same format as the Version 4 Berkeley VAX object files.

2.1 The ASIX Assembler

The ASIX assembler was developed as part of a program to upgrade the tools for the development environment for the Voice Funnel. The tools we had been using consisted of a C compiler, assembler, and linker. To this we added a range of custom programs which allowed us to manipulate the output from the MIT assembler in order to make it usable in a large multi-segment address space. This resulted in the maintenance of a large

number of accompanying software tools.

The new assembler resembles the old assembler only in that it accepts almost the same source language, so that the output of the compiler is acceptable to it. A number of enhancements have been added to the assembler in order to improve the code generated by it, provide better error detection and diagnostics, and to make the assembler easier to use. In addition, the assembler assembles programs several times as fast as the old assembler.

The output of the assembler is compatible with the Version 4 Berkeley VAX UNIX loader format, enabling us to use the Berkeley tools, usually without modification, to examine and manipulate the load files of the Butterfly development system. This enables us to take advantage of a large body of existing development tools, and to accept enhancements and extensions to these tools from future Berkeley UNIX releases with a minor effort. It also means that we are freed from software maintenance of these tools, since they are the basic system tools, maintained as part of the installation software.

The assembler has a number of interesting features to enable the assembly of more optimal code than can be provided with most assemblers. The most important of these is a span-dependent package, which permits the assembler to generate the shortest version of an instruction, based on the values of referenced

symbols. The assembler is also able to assemble together several independent routines without having local symbols with identical names in different routines clash. This is convenient not only for generating libraries, but also to permit better code generation for instructions referencing global symbols.

Other desirable features of the new assemble include the following. Symbols are divided into disjoint classes based on where they may appear syntactically; thus, symbol names will not clash with operation code names, but will clash with register names, as is desired. Registers may be given synonyms, as may all variables. Local symbols may be used which are defined only within a tightly restricted range of the program. A pseudo-operation provides a mechanism for defining symbols for the assembler's output symbol table, which are invisible to the remainder of the assembly, but which can be defined in terms of other symbols and the current assembly location counter. This allows a compiler to include symbols in the object code symbol table which may then be used as the basis of a sophisticated source-language debugger.

The addressing mode of an instruction can be specified in two ways. In the "normal" mode, it chooses the most efficient addressing mode and computes necessary addresses and offsets. The assembler also permits the explicit specification of all addressing modes. In the course of generating the most compact code possible, it generates position-independent code wherever

possible.

2.1.1 Assembler Structure

The assembler is structured around the UNIX tools lex and yacc for lexical analysis and parsing. This simplifies the generation of the assembler and also produced an assembler which is more efficient since these tools employ finite state machines for language interpretation. Additional modules are provided for the processing of operands and operators, after parsing. The assembler runs in two passes, with span-dependent processing between the passes.

The input language of the assembler is highly irregular, due both to the irregular structure of the MC68000 architecture and to the requirement for compatibility with the output language of the compiler. As a result, opcode selection and opcode-operand compatibility is handled by a large grammar in the yacc parser. Had the language been more regular, these problems would have been solved by a more table-driven approach. This means that much of the semantic processing of the assembler is built into the parser mechanism.

The semantic processing of assembler instructions depends upon the instruction encountered. Instructions take zero, one, or two operands, and each of the operands may fit general patterns of addressing modes allowed, or accept only a single

addressing mode or special register. The parser treats highly restricted operands as part of operator processing, and invokes operand processing to reduce the more general operands to an operand data structure.

The symbol table package is based on a hash-coded table, with symbols buckets in linked lists off the hash table entries. The permanent symbols (e.g. instruction mnemonics) are placed into the symbol table first, so that their lookup should seldom, if ever, require examining a second symbol. The symbol table is actually multiplexed to look like an indexed set of symbol tables. This permits permanent symbol names to be the same as labels without name conflict, and provides the ability to assemble multiple programs in one assembly without having name conflicts between programs.

2.1.2 Span-Dependent Instruction Processing

The most interesting feature of the ASIX assembler is its ability to generate instructions of different length based on the values of the operands. This is a difficult problem, since the value of an operand can only be determined in pass two of the assembler, but the uncertainty in the length of instructions in pass one makes it difficult to know the value of symbols in pass two. The solution to this problem is called "fix-up" processing, which is performed between passes one and two.

The problem has been attacked for the case of optimizing the length of jump-branch instructions for machines with short relative branches and long absolute jumps. A relative branch is an instruction which causes a jump to a location which is within a fixed range of the current program counter.

The Motorola MC68000 is even more complex. A one-word instruction (bras -- branch short) can jump within the range PC-126 to PC+129, a two-word instruction (bra -- branch) can branch within the range PC-32766 to PC+32769, and the full address address space can be reached with a three word instruction (jmp -- jump). In addition, a short (two-word) form of the jump instruction is available which will address absolute locations in the range -32768 to +32767. Many other instructions in the 68000 such as "load" and "add" have addressing modes corresponding to all but the shortest relative addressing. No previous span-dependent algorithm has been able to take advantage of the short form of the jump instruction, or has performed address optimization for any other instructions.

The span-dependent algorithm used in ASIX is an extension of an algorithm by Szymanski. * The basic feature of the algorithm is to assume in pass one that every instruction can be assembled in its shortest form, between pass one and pass two, extend the length of any instruction which has an address which cannot be

* Szymanski, Thomas G., "Assembling Code for Machines with Span-Dependent Instructions", CACM, Vol. 21, No. 4, April 1978.

reached by the form of the instruction selected, repeating until all instructions are able to be assembled with the indicated lengths, and then assemble them normally during pass two. Our extension to the Szymanski algorithm involves the inclusion of two classes of legal ranges: ranges around a location in the program (relative) and ranges of absolute numbers. The previous algorithms handled only the relative ranges.

During pass one, each instruction which may be assembled in versions of different length, either using different operand modes of the same instruction or using different MC68000 instructions, is recorded in the span-dependent data structure. This data structure includes the (symbolic) operand, and, for each version of the instruction, the length of the instruction, the eligible range, and whether the range is relative or absolute.

During pass two, the instruction is assembled in its shortest form possible, given the values of its operands, and the length selected by the span-dependent package is checked against the length of the assembled form. These should always be identical, and any difference is regarded as a fatal assembler error. Since the length generated by the span-dependent package is only used for error-checking, why does the span-dependent package need to be run at all? The answer lies in its side-effect: as it fixes the length of the instructions, it also updates the values of all the symbols in the symbol table which

are affected by the change in the length of an instruction. Without the updating of symbols, the addresses in pass two of the assembler would be wrong, and the lengths of instructions would also be wrong.

The heart of the span-dependent package is the processing which is performed between passes one and two. During this time, the data structure for each instruction is examined in turn, and, if the current length of the instruction cannot address its operand, the instruction is lengthened, and all symbols following it in the program have their values adjusted. This is repeated for each instruction. When all instructions have been thus processed, the routine starts over unless it has passed through every instruction without lengthening any. This algorithm has been shown to be NP-complete in the time required for its execution. In practice, however, it runs quickly (usually in linear time) since the data structure is unlikely to be traversed more than two or three times.

The addition of an absolute range to the span dependent package achieved two advantages: we can now make use of short absolute addressing for references to low memory addresses, and we can use the span-dependent package to select among various lengths of immediate operands and between the use of immediate operands and quick instructions which perform the same operations for operands in small ranges.

Appendix: UNIX Programmer's Manual

NAME

asix - Assembler for the M68000

SYNOPSIS

asix [-object 'oname' -listing 'lname' -trace constant] files

DESCRIPTION

ASIX assembles the specified files.

The output file is given a name constructed by appending '.o68' to the base name of the first input file, unless the -object flag is specified. If -object oname is specified, then the output file will be given the name oname. If -listing lname is specified, then a listing file will be produced with the name lname. Otherwise no listing file is produce. If -preserve is specified, then the output file will be preserved, even though errors were encountered during assembly. Otherwise the output file will be deleted if an error occurs during assembly. If -defineall is specified, then undefined symbols will be considered as globals and will not cause errors; this is the normal mode used by bfly c. If the -number switch is specified and a listing is produced, then the listing will include the line numbers from the input file(s). The -trace constant flag is included for debugging. The bits in the binary representation of the constant controll debugging features. Currently, bit zero causes the display of tokens passed between lex and yacc, and bit one provides a dump of the span-dependent package data structures.

SEE ALSO

bcc(1), b.out(5)

NAME

bcc - C compiler for the M68000

SYNOPSIS

bcc [option] ... file ...

DESCRIPTION

Bcc is a C compiler that produces code for the M68000. It accepts several types of arguments:

Arguments whose names end with '.c' or '.c68' are taken to be C source programs; they are compiled, and each object program is left on the file whose name is that of the source with '.b' substituted for '.c' (or '.c68'). The linker will normally be automatically called leaving the final linked object in the file whose name is that of the source with the suffix replaced with '.B'; The '.a68' and '.b' files are normally deleted.

In the same way, arguments whose names end with '.a68' are taken to be assembly source programs and are assembled, producing a '.b' file.

After all '.c', '.c68' and '.a68' files in the command list are dealt with, the linker will normally be invoked. All '.b' files produced will be linked together with any files in the argument list explicitly named with a '.b' suffix, archived files named with a '.a' suffix, and any libraries included by a -l request. The final output will be a file whose name is composed of the first '.c', '.c68' or '.a68' file included in the argument list, with the suffix replace with '.B'. All '.a68' and '.b' files produced as a by-product of a compilation will normally be removed after the '.B' file is produced.

If an argument consists of a file name with no '.' and no suffix, then if there is a file consisting of the supplied name followed by '.c68', it will be compiled as though its full name had been supplied; if not and there is a file consisting of the supplied name followed by '.a68', then it will be assembled as though its full name had been supplied.

A number of options are interpreted by bcc. In addition, all unrecognized items on the command line are gathered for the linker and passed to it as part of the linker command line.

-c Suppress the loading phase of the compilation. All of the '.b' temporary files will be retained, and '.b' files explicitly specified in the command string will be ignored.

-O Invoke an object-code improver.

-S Compile the named C programs, and leave the assembler-language output on corresponding files suffixed '.a68'. '.a68' and '.b' files in the command string will be ignored.

-o output
Name the final output file output. If this option is not used, the output file will be the basename of the first file in the argument list followed by '.B'.

-V Each internal operation will be printed out as it is performed.

-h The assembler will output a listing.

-B The linker will be invoked in a form suitable for stand-alone programs on the butterfly. In particular, the following linker arguments will be supplied "-T 40000 -I 50000", specifying that the text segment begins at location 0x40000 and the initialized data segment begins at 0x50000.

-sx The file /usr/butterfly/lib/bstart/startx.b where x is the string following the "-s" switch will be placed at the beginning of the link request. This option should precede any file names in the "bcc" command.

-lx The file /usr/butterfly/lib/libx.a will be included at the end of the list of file names sent to the linker.

FILES

/lib/cpp	preprocessor
/usr/butterfly/bin/bc68	compiler
/usr/butterfly/bin/bo68	optional optimizer
/usr/butterfly/bin/asix	assembler
/usr/butterfly/bin/blnk	linker
/usr/butterfly/lib	default library directory
/usr/butterfly/lib/bstart	directory for start programs

SEE ALSO

asix(I), blnk(I), bld(I)

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self-explanatory. Occasional messages may be produced by the assembler or linker.

NAME

bld - down-line loader

SYNOPSIS

bld [-- or ++] [file [termno]]

DESCRIPTION

Bld formats object files which are already linked and transmits them through an RS-232 channel to an M68000 using the USD debugger and protocol. There are three cases: (1) The load may explicitly specify a port over which to load the Butterfly. (2) The Butterfly may be loaded over standard input/output with **bld** specifying the locations for the data (**bld** master). (3) The Butterfly may be loaded over standard input/output with the Butterfly specifying the locations for the load (Butterfly master).

The first case is used for loading over a terminal port without a login shell to load the Butterfly by force-feeding the data to USD. For this case a user would specify the file to be loaded and the tty line number (**termno**) of the USD port to the M68000. If omitted, the port defaults to /dev/ttybfly, the port on which the RS232 link to the Butterfly is *normally* put. The **bld** program will then attempt to attract the attention of the M68000 and send it the file. It will print a number of indicators on standard output to indicate its progress. These include

- ***** This character is printed whenever a command line is transmitted to the M68000.
- **#** This character is printed when an initiation sequence is sent.
- **!** This character is printed when a time-out occurs on a response from the M68000. It usually indicates either a problem with the communication line or with the USD running on the M68000 (e.g., the M68000 is powered down).
- **%** This character indicates the completion of transmitting the text and data segments. Bss and a few commands to set mapping remain to be sent.

In the other two load options, the load is assumed to be initiated by invoking **bld** over the port used for loading, and this port is assumed to present a UNIX shell to the Butterfly. The second case, in which **bld** is the master, is identical in operation to the first, except that **bld** will not report the status of the load, since the standard output is used for the actual data transmission. This second case

is specified by sending the command bld with ++ as its first argument. The M68000 specifies the desired file in the file argument.

The final case (Butterfly master) differs from the second case in that the command invoking bld is assumed to be generated by a program capable of accepting an augmented command set on the Butterfly, and establishing the locations for the load. This case is specified by invoking bld with -- as its first argument. The M68000 can either specify the desired file in the file argument, or can negotiate with the bld program using the command/response modes supported by the protocol.

In actual use, the two latter modes are expected to be used as follows. The user will start up a Butterfly by attaching its host port to a VAX port with a shell on it. He will then type on the Butterfly terminal port, putting it in transparent mode to communicate with the shell. He may then type through the Butterfly to the VAX to invoke bld with the ++ option to bring in a bootstrap loader or the Chrysalis operating system. This loader or operating system will then put the host port in transparent mode and invoke bld with the -- option to load whatever processes it wishes.

Bld produces several messages describing the final status of the load, except with the -- option. These consist of the following:

aborted
: This message indicates that the M68000 has transmitted an abort sequence to terminate the transmissions. This is typically triggered by human intervention at the M68000 end.

aborting
: This message indicates an abort sequence is being sent by bld. The usual trigger is sending an interrupt signal to bld while it is transmitting.

Loaded
: This message indicates that the load sequence is complete and bld has received acknowledgement for all transmitted messages from the M68000.

SEE ALSO

asix(1), blnk(1), bcc(1)

BUGS

The scheme for force-feeding a machine is a Butterfly-specific. It sets memory mapping in a fashion suitable for

the Butterfly. The scheme also stops at down-line loading; it does not cause the program to begin execution.

NAME

bldprom - loader for prom programmmer

SYNOPSIS

bldprom file [ttylineno]

DESCRIPTION

Bldprom formats object files which are already linked (using blnk) and transmits them through an RS-232 channel to the Data I/O prom programmer using the Motorola slines format. Bldprom requires that the file to be sent be in the format used by asix and blnk with a magic number of (octal) 0407, which provides for contiguous text and data segments. This may be accomplished by providing blnk an argument of -N.

Bldprom takes two arguments, a file name containing the object code, and the number of the terminal line to send the slines output. This second parameter is the suffix of the device number for the tty port selected, XX in /dev/ttyXX. If omitted, the second argument defaults to prom, for loading over /dev/ttyprom, the device normally attached to the prom programmer.

To program a prom, first, on the Data I/O programmer, enter the key sequence SELECT 8 2 ENTER SELECT D 1 START; then, from the host machine run bldprom. Finally, on the programmer, enter PROGRAM START.

SEE ALSO

asix(1), blnk(1)

BUGS

NAME

blnk - link editor

SYNOPSIS

blnk [option] ... file ...

DESCRIPTION

Blnk combines several object programs into one, resolves external references, and searches libraries. In the simplest case several object *files* are given, and **blnk** combines them, producing an object module which can be either executed or become the input for a further **blnk** run. (In the latter case, the **-r** option must be given to preserve the relocation bits.) The output of **blnk** is left on a.out. This file is made executable only if no errors occurred during the load.

The argument routines are concatenated in the order specified. The entry point of the output is the beginning of the first routine (unless the -e option is specified).

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. If a routine from a library references another routine in the library, and the library has not been processed by **ranlib**(1), the referenced routine must appear after the referencing routine in the library. Thus the order of programs within libraries may be important. The first member of a library should be a file named `__.SYMDEF', which is understood to be a dictionary for the library as produced by **ranlib**(1); the dictionary is searched iteratively to satisfy as many references as possible.

The symbols `_etext', `_edata' and `_end' (`etext', `edata' and `end' in C) are reserved, and if referred to, are set to the first location above the program, the first location above initialized data, and the first location above all data respectively. It is erroneous to define these symbols.

Blnk understands several options. Except for -l, they should appear before the file names.

-A This option specifies incremental loading, i.e. linking is to be done in a manner so that the resulting object may be read into an already executing program. The next argument is the name of a file whose symbol table will be taken as a basis on which to define additional symbols. Only newly linked material will be entered into the text and data portions of a.out, but the new symbol table will reflect every symbol defined

before and after the incremental load. This argument must appear before any other object file in the argument list. The -T option may be used as well, and will be taken to mean that the newly linked segment will commence at the corresponding address (which must be a multiple of PAGESIZE). The default value is the old value of _end.

-D Take the next argument as a hexadecimal number and pad the data segment with zero bytes to the indicated length.

-d Force definition of common storage even if the -r flag is present.

-e The following argument is taken to be the name of the entry point of the loaded program; the beginning of the text segment is the default.

-I The next argument is a hexadecimal number which sets the data segment origin, where initialized data begins. The default origin is either immediately following the text segment or at the next PAGESIZE boundary, depending on whether the -N or -n option is in effect.

-lx This option is an abbreviation for the library name `/usr/butterfly/libx.a', where x is a string. A library is searched when its name is encountered, so the placement of a -l is significant. If x is missing, then the default library /usr/butterfly/libbs.a is used.

-M produce a primitive load map, listing the names of the files which will be loaded.

-N Do not make the text portion read only or sharable. (Use "magic number" 0407.)

-n Arrange (by giving the output file a 0410 "magic number") that when the output file is executed, the text portion will be read-only and shared among all users executing the file. This involves moving the data areas up to the first possible PAGESIZE boundary following the end of the text. This option is the default choice for blnk.

-o The name argument after -o is used as the name of the blnk output file, instead of a.out.

-P Takes the next argument as a hexadecimal number which is used as PAGESIZE for various boundary calculations.

The default value for PAGESIZE is 10000, which is the hexadecimal representation for 65536 (decimal)

-r Generate relocation bits in the output file so that it can be the subject of another blnk run. This flag also prevents final definitions from being given to common symbols, and suppresses the `undefined symbol' diagnostics.

-S `Strip' the output by removing all symbols except locals and globals.

-s `Strip' the output, that is, remove the symbol table and relocation bits to save space (but impair the usefulness of the debuggers). This information can also be removed by strip(1).

-T The next argument is a hexadecimal number which sets the text segment origin. The default origin is 0.

-t ("trace") Print the name of each file as it is processed.

-U The next argument is a hexadecimal number which sets the bss segment origin, where uninitialized data begins. The default origin is immediately following the initialized data segment.

-u Take the following argument as a symbol and enter it as undefined in the symbol table. This is useful for loading wholly from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

-X Save local symbols except for those whose names begin with `.L'. This option is used by cc(1) to discard internally-generated labels while retaining symbols local to routines.

-x Do not preserve local (non-.globl) symbols in the output symbol table; only enter external symbols. This option saves some space in the output file.

-ysym
Indicate each file in which sym appears, its type and whether the file defines or references it. Many such options may be given to trace many symbols. (It is usually necessary to begin sym with an `_', as external C, FORTRAN and Pascal variables begin with underscores.)

FILES

/usr/butterfly/lib*.a	libraries
a.out	output file

SEE ALSO

asix(1), ar(1), bcc(1), ranlib(1), bld(1)

BUGS

There is no way to force a particular data object to be page aligned.

NAME

bmake - maintain program groups for the Butterfly

SYNOPSIS

bmake [option] . . . target . . .

DESCRIPTION

Bmake is a shell script which simply runs the make(1) program to make the specified target(s). However, in addition to the "makefile" which would normally be used, make is passed the special file /usr/butterfly/bin/misc/bMakefile, which contains the additional prefixes and default rules needed to produce Butterfly software. These special rules are as follows:

```
# This file contains the appropriate `make' definitions to permit the
# building of butterfly software in addition to regular UNIX(tm)
# software, and is inserted by the `bmake' shell-file in this
# directory (ahead of the user's normal `Makefile'). Bugs to rdr...

IDIR = /usr/include /usr/butterfly/h

# Compilers and switches specific to the bfly project...

ASIXFLAGS = -def
BCC = bcc
BCFLAGS = -c -h -O
BLDFLAGS = -B -sX
BLIBES = -lbs

# The following replaces the builtin suffix list with an augmented
# version.  If the builtin list changes, this should be updated...

.SUFFIXES: .68 \
        .o68 \
        .c .c68 \
        .a68 .s68

# Rules...

.c.o68 .c68.o68:
     $(BCC) $(BCFLAGS) $<
.a68.o68 .s68.o68:
     asix $< $(ASIXFLAGS) -l $*.168
.o68.68:
     $(BCC) $(BLNKFLAGS) -o $@ $! $(BLIBES)
```

FILES

/usr/butterfly/bin/misc/bMakefile additional default rules

BUGS

Does not correctly handle the -f option.

SEE ALSO

make(1) in the normal Unix(tm) manual.

DISTRIBUTION OF THIS REPORT

Defense Advanced Research Projects Agency
Dr. Robert E. Kahn (2)
Dr. Vinton Cerf (1)

Defense Supply Service -- Washington
Jane D. Hensley (1)

Defense Documentation Center (12)

USC/ISI
Dr. Danny Cohen (2)

MIT/Lincoln Labs
Dr. Clifford J. Weinstein (3)

SRI International
Earl Craighill (1)

Rome Air Development Center
Neil Marples - RBES (1)
Julian Gitlin - DCLD (1)

Defense Communications Agency
Gino Coviello (1)

Bolt Beranek and Newman Inc.
Library
Library, Canoga Park Office
R. Bressler
R. Brooks
P. Carvey
G. Falk
J. Goodhue
E. Hahn
E. Harriman
F. Heart
M. Hoffman
M. Kraley
A. Lake
W. Mann
R. Rettberg
P. Santos
E. Starr
E. Wolf